Domobus User Interface & Example Implementation

Filipe Fernandes[1] and Tomás Jacob[2]

***Abstract*— In this document, we describe the development process, tools, mechanisms and devices used to implement a simulation of a Domobus system. Domobus is a home automation system specification to control domotics devices and other appliances. For our project we used Meteor to develop a web application that receives a Domobus XML file and allows connection with real life devices using HTTP requests.**

For the demonstration we used Raspberry Pi's and a Docker container to allow quick propagation between multiple devices. This makes possible a quick and easy integration of new devices, as well as flexibility to implement different configurations of houses, offices and other buildings.

It is possible to use our implementation as a basis for further development, both in type of available devices and interface details.

I. INTRODUCTION

House automation devices are increasingly becoming more and more common and market for these devices are increasing, with plenty of companies offering more solutions for both households, offices and other buildings. These devices range from lights and heating systems to alarms and access controls. Common people can now start creating and installing their own systems in their houses without help of specialized technicians and at affordable prices. This happens in part because these systems are made of several individual pieces and users can customize them to accurately match their needs with ease.

Amongst the benefits of systems like this are the comfort of users as well as commodity and safety. The ability of controlling devices at home while in remote places offers a whole world of possibilities. From simple tasks like turning up the heat before arriving home to more complex situations like opening a safe or a house door at a scheduled time.

With this new reality, a need for new tools and systems to help users create and implement it arises. A good example of this is the IKEA furniture, while initially people turned to it because of low pricing, right now it's also because of the easiness to transport and assemble. The assembly process is composed of specifically designed tools and rule sets to all IKEA pieces, making it accessible to every kind of client. This is the same principle we want to apply to domotics. Domobus offers a general approach to control multiple devices, but is also flexible enough to accommodate most types of devices through a modular architecture, where each new type of device is configurable on it's own. It works as a tool to put together every device necessary to the system.

Now it becomes necessary to make this systems easy to use for everyone. And most important: easy to install. On this project we develop a web application to "host" any Domobus System. The input is based on a XML file describing the building layout and devices necessary. The interface lists every device and it's properties and allows the user to control and add new devices. In our implementation, the system only has to receive the address of the device, and for as long as he is connected to the internet we can control it, by sending the system values. This way we can add new devices and we only have to set them up to process the values. The Domobus system remains unchanged and without need for new code or features.

In this particular implementation we used Raspberry Pi's as the main device to control real world equipments like lights and others. The Pi's are a relatively powerful tool, which is why we choose them. Allows for the connection of electronic components, while also running a Linux distribution. This way we can both run scripts in a variety of programming languages, process the HTTP requests and using both, control real world components. One of the features we also wanted to implement was the ability to make changes to multiple devices without much trouble. The Pi's also opened up this possibility. Using a Docker container, we are able to import centralized code and scripts to all Raspberries easily, with only two commands per Pi. A user can simply develop all necessary code centralized, deploy it to Docker and then import on each Pi, if they have an internet connection this can all be done in a single computer.

Using several tools we developed a proof of concept of a Domobus system with simple real life applications. We believe the system can be used with ease by a variety of users, and is flexible enough to be adapted to several different use cases and scenarios. The detailed working of this system is explained in the next sections.

II. MOTIVATION AND CONTEXT

A. Motivation

This project was developed as part of the Ambient Intelligence course at Instituto Superior Tcnico, under the guidance of Professor Renato Nunes. Our first idea was a replication of an office, where several users could influence and control real life appliances like ceiling lights, doors, windows and others. This would consist of simple interactions and focus more on decision making within the system. After a discussion of the initial idea with the professor we branched out a bit. Instead of an office, we decided to focus on implementing

[1]Fernandes F. (78083) is enrolled in Ambient Intelligence Course of Masters Degree in Telecommunications and Informatics Engineering, Instituto Superior Técnico, Group 06

[2]Jacob T. (78034) is enrolled in Ambient Intelligence Course of Masters Degree in Informatics and Computer Engineering, Instituto Superior Técnico, Group 06

Fig. 1. Concept Image of Home Automation

a simulation of a Domobus System and make a more home-oriented system for the front-end and implement some devices as a proof-of-concept of the whole system. This decision matched both authors field of work and personal preferences, as the first half is more design and web-oriented, and the second half relates to networks, connectivity and real-life highly scalable systems.

The web application focus is making the system work as smoothly and simply as possible, so the user can quickly turn on devices and change intensities. There is an increased focus on spatial navigation as this is the most familiar navigation for new users. A user knows his house and it's organization well, but it may not be familiar with the application, so it's important to work around the house layout first.

Regarding the second half our goal is to be able to demonstrate real actions happening. In theory we can have the perfect user interface but we want to do more and prove that this interface can control a real system and not only a simulated environment. For this we will focus on using devices and hardware that could be used in the future on a real house implementation. And also develop a system as lightweight and scalable as possible.

We hope to see this project serve at least as an example of the potentials of Domobus and the large scope of applications that can be developed using this system. While it's no more than a simulation, we believe that very few steps would still be necessary to actually implement this project in a small environment. Another use we would like to see for this project is demonstrations for schools and potential engineering students, willing to ingress in our school. We see this as a good example of the combination of areas and fields of study in our degrees.

B. Domobus

Domobus started as an academic project at Instituto Superior Técnico by Professor Renato Nunes with the goal to develop a home automation system. [1] The goals are to be a cost effective and simple solution, while maintaining flexibility to adapt to different device types and scalability to host large systems.

One of the unique aspects of the Domobus is it's architecture. It is composed of two layers: a High level and a Low level. The High level consists of supervision modules, mostly complex devices like computers and similars. These are responsible for the intelligence of the system, make calculations and/or predictions about the system, store data and other necessary informations. The Low level is made up of control modules that serve as an interface between sensors, actuators and the system itself. These are responsible for reading and writing necessary values and send them to the supervision modules (the communication between them is left to a Router module).

Another distinctive feature of the Domobus is it's scalability. While this is usually a problem with other similar technologies, the described architecture allows for a large system (hundreds of devices) without any inherent problem.

One of it's goals is also to define a generic approach for home automation. The system uses a XML file describing every aspect of the layout, such as the building it's implemented on as well as the characteristics of every device used. This way we can work with simple requests and values to make the whole system work. By sending GET and SET requests to the devices with the intended values we can develop almost everything necessary to an home automation system. The inner workings of Domobus are further explained in section V, subsection A.

III. FRAMEWORKS AND DEVICES

For this project, a large set of technologies were used, ranging from hardware to development frameworks. Each has it's own purpose and specific function in our simulation and in this section we explain how they work and why we choose them.

A. Meteor

Long gone are the days of Web 1.0, where websites consisted of HTML and a few lines of styling to be perfectly okay. With the advent of Javascript, websites started having more interaction and reactivity, instead of being just static pages, data and information change and users can perform more complex and serious tasks then just reading information and clicking buttons. For the development of these more complex web pages, frameworks started coming up. These are like tools with a set of previously defined behaviors and elements that allow for faster development. Most of these are based on some form of Javascript.

One of the most amazing feature of these frameworks is that they reduced the need for highly specialized programmers. Before, web developers divided into two groups, consisting of front-end developers and back-end developers. To put it simply, the first ones designed the website aspect and usability, while the last ones build databases, manage servers, security and performance aspects. But now, with these frameworks, a single Javascript programmer can fully construct a database and also take care of the display of those informations. The language, an extremely flexible one, allows for a programmer to write both server-side and client-side code in a single page. Meteor is one of these frameworks. It runs on Javascript, and is one of the best examples of this simultaneous development [5].

The information server side is stored on a MongoDB Database. This is an alternative to the regular mySQL databases, more commonly used. In both cases the data is persistent, but MongoDB is a non-relational database. It is also free of cost and an open-source project. One of the main reasons we used this type of database is the use of objects. The information is stored as objects, keeping related data together. This ties very closely with the XML used in Domobus, that also associates data in the same way, although this database uses JSON-like documents (check the section V, subsection A-2 on XML parsing).

For the user interface Meteor has a great advantage: Data Reactivity. On a regular web page, a request is sent to the server and the response is displayed to the user. If the information is changed on the server, the user will not see it until a new request is made, like a page refresh. The architecture on Meteor is a different paradigm know as ”full-stack reactivity”. The information is published from the server and the client signs a subscription for it, every time the server data changes the client will receive it, just like a real world newspaper subscription. This is obtained with a custom protocol knows as DDP, Distributed Data Protocol.

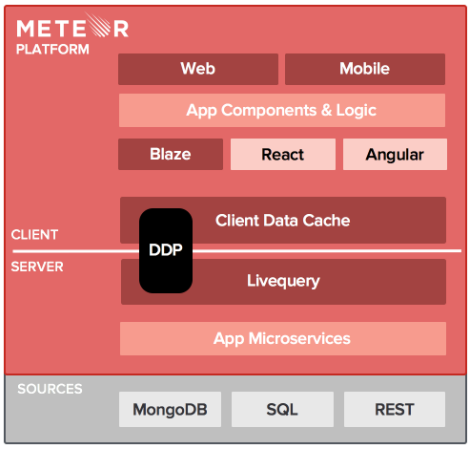

Fig. 2. Meteor Architecture

This data reactivity has a great use case in our project. The idea is for the information to be accessible to multiple users, either in a house or office building, multiple users can change the state of the devices. Meteor automatically updates the information for every user, bringing us a very precious feature without additional programing effort.

While there are some other frameworks for web development, we chose Meteor for the technical advantages it offers but also because it’s one of the easiest to use frameworks. Even with a limited previous knowledge about Javascript and web applications, Meteor empowers the developer with a bunch of tools and easy to use packages that allow very fast prototyping.

The Meteor version used for development was the 1.6.1.1 release with some additional packages for debug and some features.

B. Auxiliary Web Technologies

For a project with a focus on interface it was important to make a modern but simple design. To achieve this goal with less programming effort we used a CSS Library called Bootstrap (version 4.0-alpha-6) [7]. This library can be described as a group of CSS classes already styled and ready to use. We some personal customization, it can be used to quickly design a user interface, and reuse code. One of the most important features of this library is the grid layout. Bootstrap has a set of predefined classes that create a grid of rows and columns to place our information into and assign colors and styles with ease.

Another regular presence in modern web applications is jQuery. This is another library, this time based on Javascript, also a free and open-source project. It’s purpose is to manipulate HTML elements, with several useful previously defined functions for it. As a complement, jQuery UI, one of it’s child projects, was also used in our project.

C. HTTP Requests and JSON and XML

Meteor handles the client and server communication for the web application, but another important element for the system is the communication between the web server and the devices. This is achieved with regular HTTP requests, sent straight from the Meteor server and received on each Raspberry Pi that is running the Flask software explained further below.

Another aspect mentioned before is the XML and JSON formats. XML, Extensible Markup Language, is a markup language to encode documents for both humans and machines. This is the choice for Domobus systems descriptions, but the MongoDB database works with JSON objects. JSON, JavaScript Object Notation, has a similar purpose as XML but a different way to organize the objects. This was one of the hardships we faced for the implementation (check the section V, subsection A-1).

D. Raspberry Pi

The Raspberry Pi [4] is sometimes described as a ”credit card sized computer” and we think that description is very suitable. These devices were created with a more educational objective but soon took the over Internet of Things world with it’s very affordable price and commodity of use.

These are literally small computers, it’s earliest model (Raspberry Pi Model B) has the same computational power as a 1999 Pentium II and a graphics capacity similar to an original XBox and came equipped with 256MB of RAM memory.

All of this is normally used with a specific operative system, commonly Raspbian, a Debian distribution. This provides every basic function for the Raspberry to work.

Over the years, the newest models have been upgraded not only with better CPU’s and GPU’s but also a lot of connectivity. Raspberry pi 3 B+ has 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE and its still credit card sized.

The main reasons that led us to use this devices were precisely its huge connectivity and compatibility to electronic components like leds and sensors but also our easy access to them. Besides that, and as we will explain on the next subsections is also important that with a few tools, a running Raspberry environment can be easily replicated.

E. Flask

Flask is a micro framework for *Python*. It provides a built-in development server and debugger, RESTful request dispatching.

It allows us to run a lightweight server responsible for providing the gateway between the user interface and actions executed on the Raspberry Pi.

We decided to use Flask firstly because it is a Python framework, the perfect language to run on Raspberry Pi and finally because it's very lightweight and easily combined with other Python modules to control sensors connected to the device itself.

F. Docker

Docker [2] is a widely used container platform for system administrators and developers. It is used to develop, run and deploy applications with containers. It runs on almost every architecture, including *ARM* (raspberry pi architecture).

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.

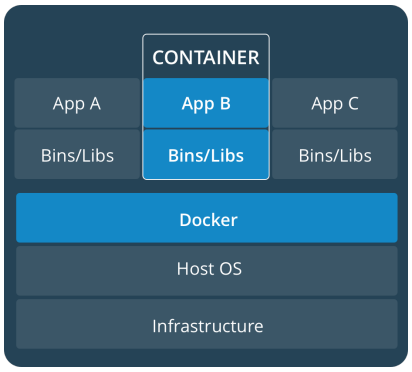

Fig. 3. Docker Container

Containers are widely used nowadays because they are:

- Flexible
  Every application can be containerized
- Lightweight
  Containers leverage and share the host kernel
- Interchangeable
  It's possible deploy updates and upgrades on-the-fly.
- Portable
  They can be built locally, deploy to the cloud, and run anywhere.
- Scalable
  Containers can be automatically replicated and distributed
- Stackable
  Services can be stackable vertically and on-the-fly

By using Docker containerization we can be sure that both our development and production environments are the same, this combined with its huge scalability makes Docker the perfect platform for us to use. Thanks to Docker after building our container image and setup the hardware, running our software is as easy as running two commands.

Containers are commonly compared to virtual machines when a subject is keeping the same production and develop environment. But the one of the docker (container) advantages and the most important to us is the containers lightweight property.

G. MPD and MPC

Music Player Daemon and Music Player Client are softwares to reproduce music. Both runs on raspberry pi and are fully controlled via command line. Combined with some streams and proper sound system allows us to provide a radio streaming service. Once again one of the reasons we opted for this tool is its lightweight property.

H. Electronic Components

For this project we used:

- 4 Raspberry Pi Model B
- 4 Led lights
- 3 Breadboards
- 4 330 R resistors
- DS18B20 - One Wire Digital Temperature Sensor
- 1 47K resistor
- Some jumper wires
- 5 RJ45 Ethernet cables
- 1 Switch
- 4 Gb SD cards
- 4 5V Power Adapter
- 1 Speaker system

IV. System Architecture

Our solution is divided in two sections. We will refer to them as the High and Low level modules. The high level module is composed by the web application and respective user interface, the low level module is composed by several Raspberry Pi's and respective hardware and electronic components.

A global view of the architecture is shown on Figure 4, and explained in detail further below.

A. Web Application

Domobus has a series of papers describing it's implementation and specifications, but there are only some drafts with suggestions for a user interface, so we are looking here at a completely new interface design. This gave us some freedom in terms of design choices and options, but also the responsibility of proving that it's possible to create a user-friendly system. As in all interface designs, the first proposed solution was not the final and even the presented solution is open to new improvements based on user feedback.

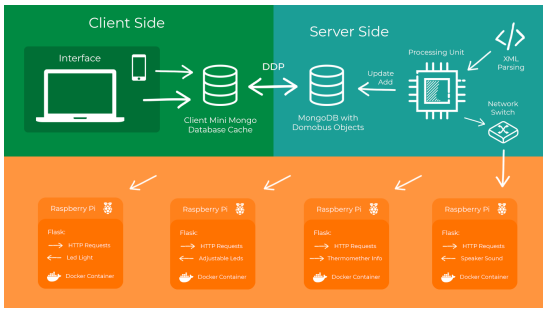

Fig. 4. System Architecture

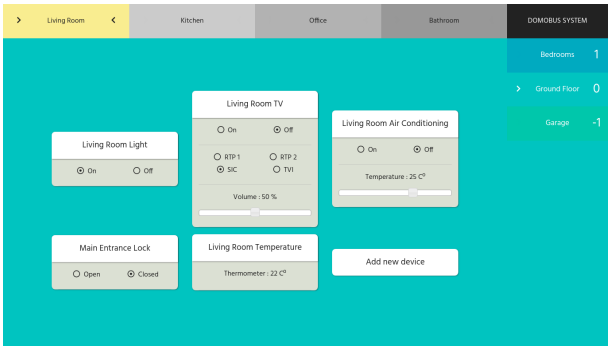

Fig. 5. User Interface on Mobile Device

We had one element in mind for the first designs, we wanted to make the interface focus on the devices. These should be the most important element in the screen, and every other element is there to assist. But the first realization we had in the drawing board is that, while this sounds good for one or two devices, it doesn't work well for a regular sized house. At this point, we decided that some investigation was needed before a more serious design.

1) User Analysis: We choose to make some user analysis to better understand the needs of potential users of the system. Home owners are the first potential user group and our main focus. This investigation was conducted through questionnaires and some interviews with 32 respondents. The first conclusion was that 68% of our users are amongst the 35-55 age group and of this group 60% are also not familiarized with home automation systems. This raised an alert interface-wise: we could not force a modern design and had to bear in mind the necessities of an older population using new technologies, such as high contrast colors and familiar buttons.

This user analysis was not conducted extensively and with a small user pool, as this was not the exact scope of the project, but to prevent design errors at least some analysis was required. These results can be improved with new tests.

With this information in mind, we still decided to make the devices the center piece, but with a simpler approach, where each device would present to the user as a card (check Figure 8), with it's properties within. While this sounds good in paper, our first paper prototype test revealed that a simple list of devices made it almost impossible for the user to identify devices and quickly navigate through them.

The proposed iteration was a spatial navigation between devices. Using the Domobus properties, each device is encapsulated within it's division, and each division within a floor. This way, users can navigate in floors and divisions and quickly see relevant devices. Our tests to this iteration revealed an unexpected consequence, when users had a house layout similar to their homes, they could find target devices much faster.

While we may think of home automation devices as complex devices, the truth is that more and more of these devices have rather simple interfaces, like on and off switches and sliders. This is very well explored in newer devices that are almost always accompanied by a mobile application to control them. There is nothing wrong with this approach, but we fell that forcing a user to install applications and such can be a turn-off for some people, therefore, we always had the idea to make our web application mobile friendly. This is not a native iOs or Android application but instead a simple web page with some custom elements when accessed on a mobile device. The previously mentioned Bootstrap library was a great helper for this feature.

So for the mobile interface we kept the focus on devices but some navigation elements had to be rearranged to fit mobile displays. One of the main differences is on the display of the device cards. While the desktop version lists devices side by side, the mobile had a scroll approach, a motion users are already familiar with on their smart phones (check Figure 6).

More user tests are available in section X after the implementation of the solution to assert it's effectiveness.

Besides the user interface, the web application is still comprised of the web server. This is implemented using the Meteor framework described previously, and details of this implementation are available on section X. This framework allows us for data reactivity, meaning every user has the information available right after changes are made to the server, without need for a page refresh. This is a very valuable feature because multiple users may use the system simultaneously without interference. This is very important for potential implementations of this system in office buildings.

The web server has another important function: the parsing of information. Domobus systems are described with a XML file detailing the building layout, devices and their properties. So one of the server functions is parsing this file into actual data the interface web pages can read. While the conversion from XML to other formats like JSON is easy and available online, processing these formats to actual objects in the system was a complex task and required a tailored parser.

This conversion from XML to objects was done using the Domobus XML Specifications to make the system as flexible as possible. This way, on paper, every XML can be processed by the application and display the information accordingly. For every update to the Specifications the parser will need to be updated as well.

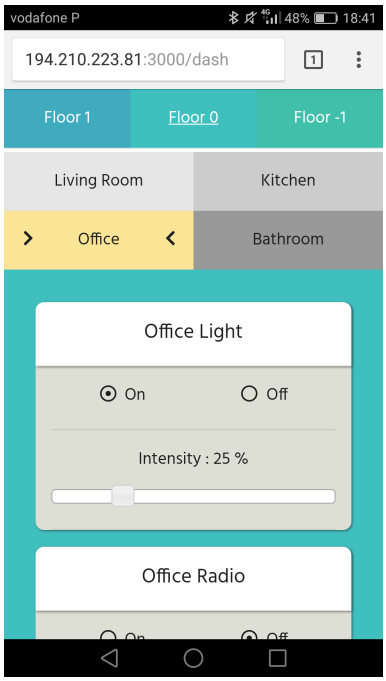

Fig. 6. User Interface Iteration

B. Low level module

Being a real device implementation the goal of this module is to communicate with the layer/module above and manage/-control sensors and devices. The upper layer of this module (low level) provides a gateway for the higher level module to communicate with this one. This gateway will be explained in detail on the next subsection.

The main hardware components of this module are Raspberry Pi's, all of them running exactly the same Linux distribution and software (this is possible through the Docker container), but all of them are connect to different electronic components.

1) Hardware: Common connections:

All of the Raspberry's share the same network connection via *eth0* interface using an *ethernet* switch and all are powered by +5.1V micro USB supply.

Raspberry Pi #1 additional connection:

- 3 leds via GPIO pins 18, 17, 27

Raspberry Pi #2 additional connection:

- DS18B20 One wire Digital Temperature sensor via GPIO output.

Raspberry Pi #3 additional connection:

- 1 led via GPIO pin 18

Raspberry Pi #4 additional connection:

- Speaker system via 3,5 mm audio jack

2) Software: All Raspberry Pi modules are running the same Linux distribution, Raspbian Stretch (*debian* based *distro*). Each one of these runs the same docker container (previously prepared), the base operative system of these containers are Raspbian Jessie, the recommended Docker base image for Raspberry Pi at the time of writing this document. This container includes all the necessary files and software to run the desired application.

Our application is composed by a Flask (python) server, led lights, temperature and music modules. Since all of the tools we used are lightweight combining all of these modules on the same image container only brings advantages to our solution.

Each raspberry has a 8Gb SD card and the compressed container only has about 200Mb so by combining all of the modules on the same container we can easily change the function of each Raspberry. It is only required to "attach" different hardware, the software is already prepared to handle several different devices.

When a container start it execute *app.py* this files starts the flask server that handles HTTP requests. More details on this will be explained in the next section.

The led lights, temperature and music modules sets up the required initializations to control the respective functions.

Linux package dependencies of this module:

- python
- python2.7-dev
- python-pip
- python-dev
- gcc
- libraspberrypi0 libraspberrypi-dev libraspberrypi-doc libraspberrypi-bin
- mpd mpc

Python package dependencies:

- click==6.7
- Flask==0.12.2
- Flask-HTTPAuth==3.2.3
- freeze==1.0.10
- itsdangerous==0.24
- Jinja2==2.10
- MarkupSafe==1.0
- six==1.11.0
- Werkzeug==0.14.1
- wheel
- RPi.GPIO

C. Interoperability/connectivity between modules

As explained previously, the Raspberry Pi modules make up the Low level module and also act as a gateway to provide interoperability between the two modules (High and Low level).

To do that, Raspberry is running a Flask application exposed on port 5000 that listens to the following HTTP requests:

- Control single led on/off — http://device_addr:5000/on?pin=18
- Control intensity leds — http://device_addr:5000/intensity?val=0
- Setup radio stream — http://device_addr:5000/radio
- Play current stream — http://device_addr:5000/play
- Stop current stream — http://device_addr:5000/stop
- Set volume — http://device_addr:5000/volume?val=75

- Get state of specified led — http://device_addr:5000/state?pin=27

Snippet of code that listens to intensity led control request and performs the respective action:

```
HTTP request example:
http://device\_address:5000/intensity?val=2

Respective code:

@app.route('/intensity', methods=['GET'])
def l_int():
        val = int(request.args.get('val'))
    if(val==1):
                led_on(18)
                led_off(17)
                led_off(27)
                return ("1")
        elif(val==2):
                led_on(18)
                led_on(17)
                led_off(27)
                return ("2")
        elif(val==3):
                led_on(18)
                led_on(17)
                led_on(27)
                return ("3")
        else:
                led_off(18)
                led_off(17)
                led_off(27)
        return ("0")
```

V. Implementation

A. Web Application

The whole application was built using the Meteor framework. As introduced before, Meteor is a Javascript full-stack framework, this means that most code was written in Javascript, with some front-end components using HTML and most styling being done in CSS. This framework handles a lot of the complex parts of settings up a server. It requires no configuration to run locally, all we have to do is add features. For our simulation we are indeed running the server on a regular computer, the only requirement is for the external devices to be in the same network as the server.

But even so, our application is made of some different parts working together. We can divide them into three main functionalities: Database Schemes, XML Parser and User Interface.

1) Database Schemes: Domobus exists as a way to generalize home automation devices, and for this goal, it sets a series of objects that can be adapted and incorporated to represent a system. Our database was made to reflect, as accurately as possible, the main features of the DomoBus. [3]

One of the first specifications of the systems is the type of properties devices can hold. The approach taken was that nearly every property can be fitted into one of two cases: Scalar or Enum. Scalar values go from a minimum to a maximum value in steps. Enum values are pairs name-value. With these two types, we can set a series of property types, such as:

- On-Off
  a pair where On equals 1 and Off equals 0
- Low-Medium-High
  an enumerated type, where Low equals 0, Medium equals 1 and High equals 2
- Intensity
  a scalar type, ranging from 0 to 100 %, with a step of 10%
- Temperature
  a scalar type, ranging from -55 to 125 C, with a step of 1 C

With these two types we can set up nearly every property a system like this needs. Scalar Value Types also use a field for it's unit as well as the number of bits necessary. Each one of these types is identified with an ID, that we can use to reference it in other objects. For the Javascript implementation, we just need to set up a Collection Schema with the parameters we need. These are all stored in JSON objects in the Mongo database, with the Temperature example translating into this:

```
{
    "ID": 1,
    "name": "Temperature",
    "numbits": 8,
    "units": "C",
    "minValue": -55,
    "maxValue": 125,
    "step": 1
}
```

After setting up the different Property Types, we set up an actual Property object. Our properties, per Domobus specifications, have only one new parameter, called Access Mode. This is one of three states, either "Write Only"(WO), "Read Only"(RO) or "Read and Write" (RW). A Property object also has a field called ValueType and RefValueType. These fields are exactly to reference the previous properties. For example, to define a Thermometer, we would now create an object with a ValueType of Scalar and a RefValueType of 1. This would indicate the property type of type Scalar with ID 1 is incorporated into our new Property.

But if we think back of real life appliances, most of them have multiple properties. We can think of a Television, besides being on or off, we also have another properties like the channel and volume. With this, we have one of the most important objects of our database: the Device Type. Each device type has an ID, a name and description and a list of it's properties and each property refers to a property value. This way, with chained objects we can grab a Device Type object and have access to all it's informations. A possible implementation of a TV can be achieved with the following JSON object, where we can already see some other properties like Volume and Channel:

```
{
   "ID": 6,
   "name": "TV",
   "description": "Television",
   "propertyList": [
    {
       "ID": 1,
```

```
        "name": "On-Off",
        "accessMode": "RW",
        "valueType": "ENUM",
        "refValueType": 1
    },
    {
        "ID": 2,
        "name": "Canal",
        "accessMode": "RW",
        "valueType": "ENUM",
        "refValueType": 2
    },
    {
        "ID": 3,
        "name": "Volume",
        "accessMode": "RW",
        "valueType": "SCALAR",
        "refValueType": 5
    }
  ]
}
```

At this point, we have the concept of device implemented in our Schemas and can create objects to match these type of devices. But it is necessary to also build similar objects for the house layout. The House structure follow a simpler line, where a House contains floors and floors contains divisions. A House JSON object goes as follows:

```
{
    "ID": 1,
    "name": "SampleHouse",
    "address": "ExampleStreet",
    "phone": "123",
    "floors": [
        1,
        2
    ],
    "divisions": [
        1,
        2,
        3
    ]
}
```

We can see the House contains the ID's of it's floors and divisions. But now, if we access a Division object, we can see that it also refers the Floor it is associated with:

```
{
   "ID": 3,
   "name": "Kitchen",
   "refFloor": 1
}
```

This way each division knows each floor it is on, and the house knows all divisions and floors it contains. There is one aditional field on each Floor called Height Order. This allows the system to iterate the floors from the ground up.

With the house structure and the device structure assembled, we are only missing the Devices themselves. While we defined the type of devices that exist in our system, we also need to define these devices (ex: more than one TV per house). These Device objects work as instances of our devices, they have an ID, a name, an associated division, an address field we will use for the communication and a Ref Device Type field where we state which of the previously defined device type this specific device is.

```
{
    "ID": 8,
    "name": "Fridge",
    "refDeviceType": 5,
    "address": "http://10.123.123.123/",
    "refDivision": 3
}
```

With this object we know there is a device of type 5 in the division 3. With access to all the objects we can see which type is number 5 and find out it's a Fridge, with On/Off and Temperature properties.

These objects allow for a definition of a Domobus System. It's possible to create almost every common household appliance and other smart devices. This is one of the most impressive features of Domobus, with only a few objects it's possible to recreate an almost infinity number of different types of devices.

The system itself, using the GET and SET requests can exist with these objects, but if we want to hold data somewhere other then the devices we need a last type of object, called Device State. This is a simple object, where we just need to know, which device and property we are referring to and it's value.

```
{
   "refDevice": 1,
   "refProperty": 1,
   "value": 0,
   "invalidValue": false
}
```

With this simple "trick" we can now hold the state of our devices, display them to the user and also update this value if necessary, either by user update or device update.

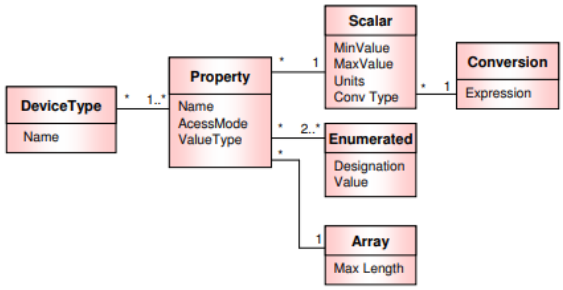

Fig. 7. Domobus Objects UML

2) XML Parser: All these objects explained before are part of the definition of a Domobus System, but, as a way to keep this objects similar across systems, XML rules were implemented. These rules are meant to guide computers to process the information and create the necessary objects to recreate the system. To accurately read these files into our system, we created a parser of XML files into MongoDB objects. This parser iterates over the different XML tags in the file, and creates the objects accordingly.

For our implementation we get the XML through an HTTP Request, this way we can change the layout of our implementation to any other.

3) User Interface: An aspect of the user interface unmentioned yet was the use of a single page. Many modern websites try to display all the information within one page to expedite user interactions. We wanted to display every action on one page as well, but with the goal to create an

easy navigation. There is no need to create several pages and layouts if the user spends most of it's time in a single page. So the goal was to give the user a dashboard to control as much devices as relevant. For the simulation we also added a simple landing page to allow loading of the XML.

The Meteor framework uses BlazeJS for it's front-end interaction. This is a template based system to allow reuse of code. If there are identical bits it allows passage of parameters to a template and it is rendered as many times as necessary. Our implementation is a great use case for this feature, because there is a constant repetition of elements. Devices are similar in terms of design, and can be defined as a template and reused. The following code is an example of this:

```
{{#each device rD}}
  <div class="col-12 device-card" >
    <span class="fs-3">{{name}}</span>
  </div>
{{/each}}
```

For each device the server communicates to the client, this bit of code is repeated. This feature is called Spacebars and is one important piece of our dashboard structure. It's used for the devices, the divisions and floors. The Javascript supporting this execution goes as follows:

```
device(rD) {
  return Device.find({"refDivision":rD});
}
```

Here we can see the client is finding every device in a specific division, by requesting only objects whose division parameter matches the one we want, this way we can set up our navigation by requesting only the devices for each area the user selects. This is a small sample of how the user interface was implemented after the design stage.

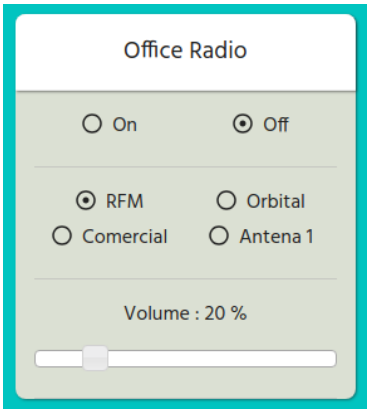

Fig. 8. Device Card

B. Docker container

As stated above we use Docker containers so we can ensure that every device in production is running the same environment of the development this way we can prevent bugs and malfunctions.

A container is the runtime of a previously prepared image. And this image is built from a text file with instructions, this file is called *Dockerfile*. It must have every necessary information to create an image of a working container. From operating system to software and its dependencies along with commands to run on the environment before running the desired application.

Below there is a snippet of code with our *Dockerfile*.

```
FROM resin/rpi-raspbian

RUN apt-get update && apt-get install -y \
python \
python2.7-dev \
python-pip \
python-dev \
gcc \
libraspberrypi0 libraspberrypi-dev \
    libraspberrypi-doc libraspberrypi-bin \
mpd mpc

ADD ./requirements.txt ./requirements.txt

RUN pip install -r requirements.txt

ADD ./app.py ./app.py
ADD ./led2.py ./led2.py
ADD ./temp2.py ./temp2.py
ADD ./music/m1.mp3 /var/lib/mpd/music/m1.mp3

ENTRYPOINT service mpd restart &&
mpc add http://centova.radios.pt:8401/stream.mp3/1
&& python app.py

EXPOSE 5000
```

The image that results from this file were uploaded to *docker* cloud to facilitate its use as we will explain on the next subsection.

C. External Devices

The first thing we did to implement our system were to flash our SD cards with Raspbian Stretch, a linux distribution based on Debian.

After booting each raspberry from an SD card it is required to setup some common operating system configuration like language, locale and internet access settings. Then it is necessary to install and setup Docker. After that two commands are enough to deploy our system:

```
- Install Docker on raspberry pi:

```
$ curl -sSL https://get.docker.com | sh
```
- Start docker:
```
$ sudo systemctl start docker
```
- Add your user (pi) to docker group
```
$ sudo usermod -aG docker pi
```

- Download docker image
```
$ docker pull filiperfernandes/ai-domotics:v3
```

- Run it:
```
$ docker run -ti --privileged -p 5000:5000
--rm -v /dev/snd:/dev/snd
filiperfernandes/ai-domotics:v3
```
```

After running *docker run..* a docker container will start and run our raspberry application.

This application can be divided into three modules. The *Flask REST API*, the led light control module and temperature measurement module. Each one of these modules is a different python file (the ones added at the end of dockerfile), *app.py, led2.py and temp2.py*, respectively.

The led module contains the initial setup and functions to control leds via GPIO output. Below are sample led On and Off functions:

```
def led_on(pin):

        print ("LED on")
        GPIO.output(pin,GPIO.HIGH)

def led_off(pin):

        print ("LED off")
        GPIO.output(pin,GPIO.LOW)
```

The temperature module as it states contains the necessary setup and functions to measure real live temperature using DS18B20 sensor.

Finally, the application itself imports both of these modules and runs a python web server using *Flask* (as explained in previous sections).

From now on let's assume that each Raspberry is a bundle of the Raspberry itself, respective SD card with *Raspbian* and all necessary software installed and configured, *ethernet* connectivity to a *localhost* network and respective power supply.

For this implementation each Raspberry has it's own purpose. One to control a simple on/off led, one to control 3 led lights to simulate an intensity value, one to measure the temperature using the DS18B20 and the last one to control a radio streaming service.

1) Intensity light: The first Raspberry is used to simulate a intensity light on the Office division, since all of our led lights are on/off led we use a bundle of three to simulate different intensities, 1 to 3 (dimmer to brighter).

This module is composed by the Raspberry Pi, 1 breadboard, 3 led lights, 3 330R resistor, and 6 M/F jumper wires.

Led lights 1, 2 and 3 are powered by GPIO pins 18, 17, 27 respectively. Each led is also connected to a ground GPIO pin passing first through the resistor.

2) Temperature sensor: The second Raspberry is used to measure the temperature of the living room. For this module we used the Raspberry bundle along with a breadboard, the DS18B20 one wire digital temperature sensor, 47K resistor and 10 jumper wires.

The third Raspberry Pi module is a simple on/off led in the kitchen, it is composed by 1 breadboard, 1 led and 1 330 R resistor.

3) Radio: The fourth and last module is a radio streaming service on the office, composed by 1 raspberry pi bundle and 1 speaker system.

Figure 9 shows one of our setups to develop the system.

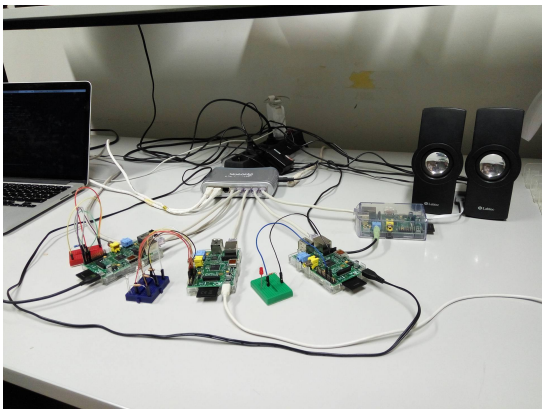

Fig. 9. Simulation Setup

VI. Evaluation

Our implementation can be divided in two parts, user interface and hardware components/actions.

In order to test its interoperability all raspberry modules and a PC hosting the user interface program were connected to the same network via *ethernet* cables using a switch.

A. User Interface

After the design and implementation of the interface, we did some user testing to prove the design was accessible and effective. Due to time constrains, this user testing was not extensive and there is room for improvements and more iterations of the design cycle.

The first batch of tests was conducted in a controlled environment and with 7 users within the same age group as our previous investigation recommended. Our test consisted of two tasks:

- "Change the Office Radio station to Antena 1"
  All users could finish this task and only one made mistakes.
  The average time to finish the task was 23 seconds.
- "Add a device called Floor Light with the address http://10.123.123.123/ and of type Light to the Master Bedroom"
  50% of the users finished the task without mistakes.
  Of the remaining users, all had some sort of problem with the address system.
  The average time to finish the task was 1 minute and 4 seconds.

This brief user testing mostly revealed that the interface works as intended and users don't have problems controlling the system. But some hardships arise in terms of configuration. When adding a device, users add some problem with the terminology and knowing what each field actually required.

For the next iterations

B. Devices specifications

Network: 192.168.1.0/24

Host PC:

IP Address: 192.168.1.1

Running software/function: Meteor application with user interface

Raspberry pi #1:

IP Address: 192.168.1.2

Simulated house division: Office

Running software/function: Intensity led control

HTTP request(s):

http://192.168.1.2:5000/intensity?val=2

Raspberry pi #2:

IP Address: 192.168.1.3

Simulated house division: Office

Running software/function: Temperature measurement

HTTP request(s):

http://192.168.1.3:5000/temp

Raspberry pi #3:

IP Address: 192.168.1.4

Simulated house division: Kitchen

Running software/function: Single on/off led control

HTTP request(s):

http://192.168.1.4:5000/on?pin=18

Raspberry pi #4:

IP Address: 192.168.1.5

Simulated house division: Living room

Running software/function: Radio streaming service

HTTP request(s):

http://192.168.1.5:5000/play

http://192.168.1.5:5000/stop

http://192.168.1.5:5000/radio

http://192.168.1.5:5000/volume?val=75

C. Devices testing

We performed an exhaustive testing regarding the Low Level module and its connectivity with the Hight level one.

First the whole set of available HTTP requests were made using *Postman*, a software widely used for API development and testing.

All of the responses obtained from the request were the expected except for the measured temperature, that always returned values higher than the current temperature(26C instead of 22C). We suspect that this happens because the breadboard that we are using is very small and the temperature sensor is close to the resistor and the other wires.

To confirm our hypothesis we approached cold objects to the sensor and the measured temperature immediately started to slowly drop (from 26C to 19C).

VII. FUTURE WORK

There are several areas of this project that can be improved and further features implemented.

A. Domobus

Our implementation of the Domobus features was limited to the fundamental objects for a working simulation. It's possible to work with the devices and their properties, but there are some other features left for implementation on future versions of this project. User definitions and access control is the most "urgent" to properly allow the system to work with multiple users, taking full effect of the features offered by Meteor. Afterwards, services, device classes and conversions should be implemented to allow manipulation of the devices and other functionalities. Regarding the user interface, the priority should be the implementation of favorite and scenarios system, to facilitate the user interaction, these allow faster navigation and selection of relevant devices.

B. User Interface

Every user interface needs some level of testing and investigation, and this project was not an exception. But we have to recognize it was not our main focus. We wanted to obtain a simple and functional proof that Domobus can be used with a Web Application to control real world devices. So one of the developments we see necessary is more user tests and, if possible, tests in actual smart home environment, to prevent possible implementation design errors.

C. External Devices

In our simulation, as explained previously, we choose Raspberry Pi's as quick and scalable way to create new devices to integrate with the application. One are of possible investigations is the compability of other devices like Arduino or similar. Another interesting investigation would be which Raspberry Model would be more suited for each device. There are around 5 different models and while some devices like a TV may require a more potent device, light switches and such could be implemented with some smaller models.

VIII. CONCLUSIONS

Domotics and smart devices are a market now starting to get explored by the masses and general public and this creates a need for better tools so every person, regardless of technical expertise, can install and customize these devices.

Our project intends to prove that this goal is not far from reach and that with the right software and hardware, applications can be developed quickly and with the use of the specification models like Domobus they can also be flexible and scalable to use in large systems.

The implementation we achieved in this project shows exactly this. Using a full-stack development framework and Raspberry Pi's we connected a series of devices that can be controlled by multiple users simultaneously and scale well enough to strive.

APPENDIX

User Interface Dashboard,

showing the devices available at the Living Room in Ground Floor

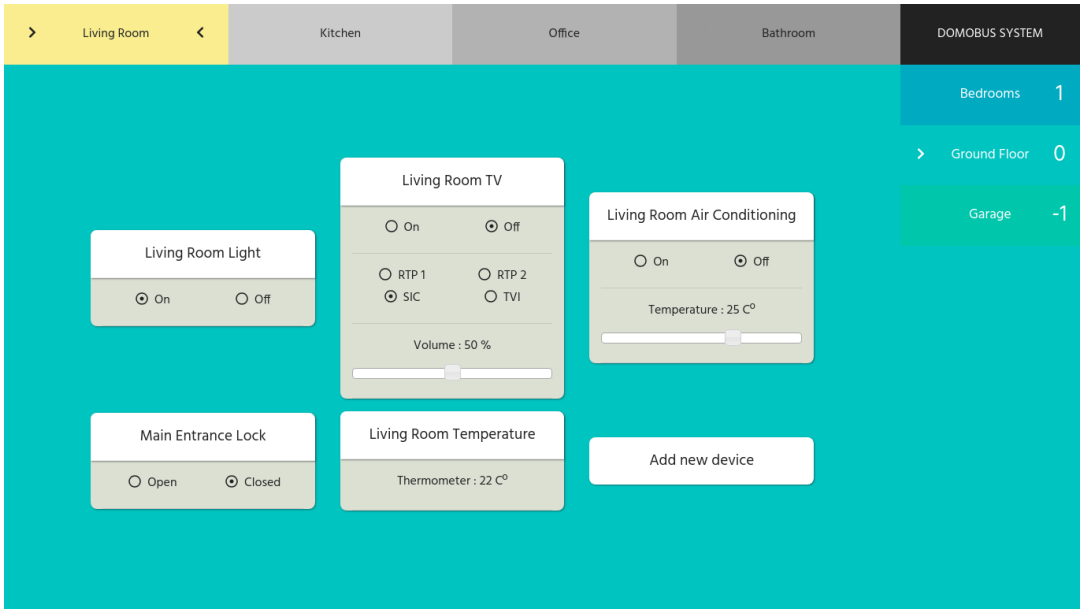

Fig. 10. User Interface Dashboard

ACKNOWLEDGMENT

The authors would like to thank to the professors of Ambient Intelligence course, Professor Renato Nunes and Professor Alberto Cunha for the guidance during the semester.

In this course we had the opportunity to mix multiple knowledge from different areas from designing user interfaces to setup electrical and electronic components passing through a simple network configuration and database administration.

We would also like to thanks NEETI-IST(Núcleo de Estudantes de Engenharia de Telecomunicações e Informática) for borrowing us all of the devices and components we needed to develop our solution and also for letting us use their room to work and setup our solution.

REFERENCES

[1] Nunes, Renato, DomoBus -A New Approach to Home Automation
[2] Merkel, Dirk, Docker: Lightweight Linux Containers for Consistent Development and Deployment, 2014
[3] Nunes, Renato, Modelo de Especificao e Programao de um Sistema Domtico
[4] Raspberry Pi Foundation, https://www.raspberrypi.org/ ; 2018
[5] Meteor Development Group Inc. http://meteor.com ; 2018
[6] MongoDB, Inc. https://www.mongodb.com/ ; 2018
[7] Bootstrap by @mdo and @fat, https://v4-alpha.getbootstrap.com/getting-started/introduction/ ; 2018